

Ask for Answer about C++ Exception Handling with DWARF (Mitigation of Memory Corruption Attacks)

Masahiro YOKOYAMA, Takamichi SAITO (Meiji University),
Kuniyasu SUZAKI (AIST)

ご挨拶

- ・我々は、プログラムの脆弱性及び脆弱性を悪用する攻撃への対策として、プログラムローダを用いて、脆弱性を招くライブラリ関数を安全な関数に置換する手法を研究しています
 - ・本日は,
 1. 開発中のローダの概要
(今年の1月末に国内の学会で発表したものになります)
 2. 現状の課題 (C++の例外処理に関して)
についてお話させていただきます
- 課題を解決するための知見をいただければと考えています

プログラムローダを用いた関数の置換により Stack-based Buffer Overflow攻撃を緩和する 手法の提案と実装

齋藤 孝道* 横山 雅展* 王 氷[†] 宮崎 博行[†]

近藤 秀太[†] 渡辺 亮平[†] 菅原 捷汰*

明治大学* 明治大学大学院[†]

はじめに

Stack-based Buffer Overflow (SBoF) 脆弱性

- ・ CやC++で作成されたプログラムにおいて、スタック上に確保されたバッファにその容量を上回るサイズの入力を許してしまう脆弱性
- ・ CWE-121に分類

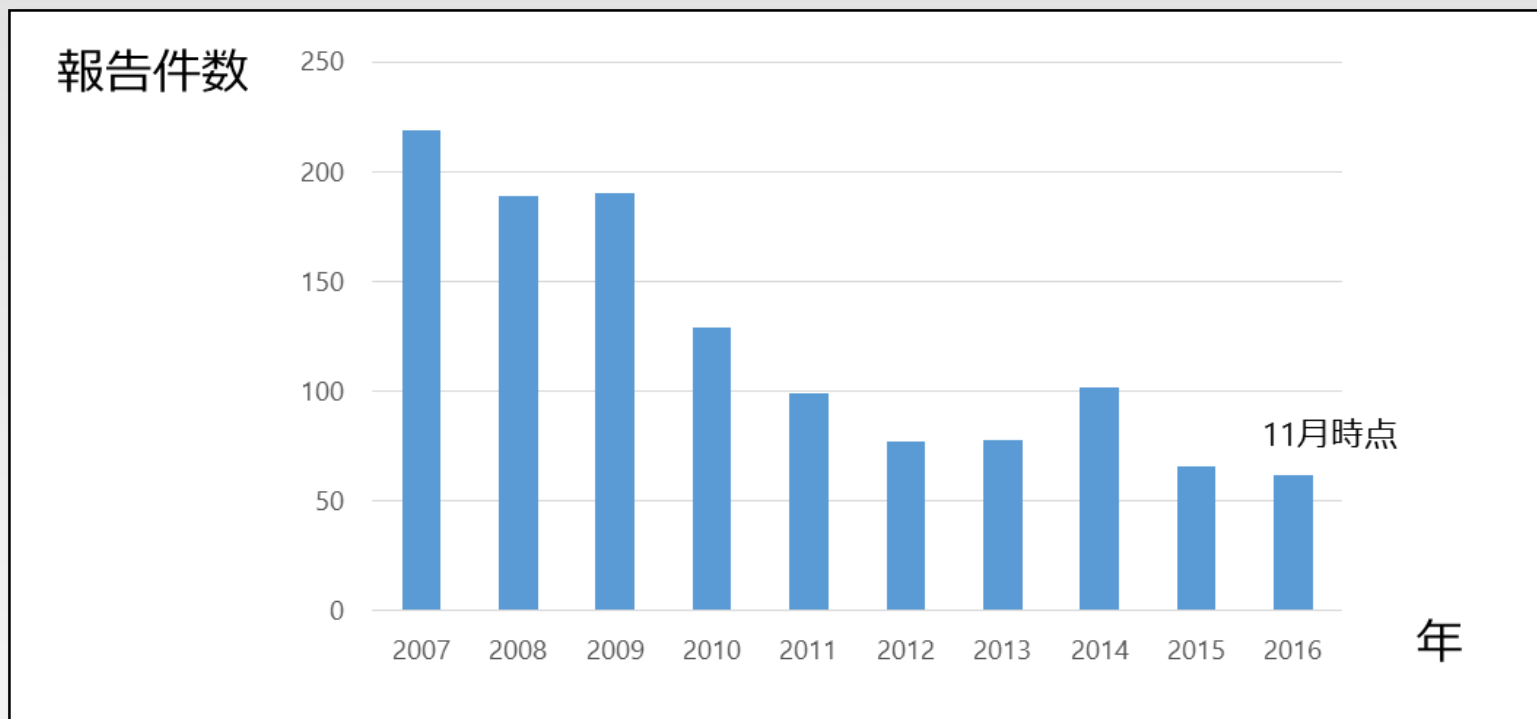
SBoF攻撃

- ・ SBoF脆弱性を悪用し、リターンアドレスなどスタック上に配置されるデータを書き換える攻撃
- ・ 想定される**脅威**
 - プログラムの実行のクラッシュ
 - 端末の制御の奪取

研究背景(1/3)

SBoF脆弱性の報告件数(NVDより)

減少傾向にあったが、2012年以降一定数で推移



⇒ SBoF攻撃の対策は**依然として重要**

研究背景(2/3)

- ・ バイナリにSBoF脆弱性が作り込まれる一因
 - 適切な書き込み先のバッファの境界検査を行わず, **SBoF脆弱性を招くライブラリ関数**を利用していること

例) **strcpy関数, strcat関数**など ...
(関数内に書き込み先のバッファの境界検査処理を含まない)

- ・ 調査研究により, こうした関数を利用するバイナリが既に一定数配布されていることが判明

研究背景(3/3)

既存の対策技術と問題点

コンパイラ

StackShield
[2000]

RAD
[2001]

SafeSEH
[2003]

GS
[2003]

Exec Stack
[2003]

PIE
[2004]

RELRO
[2004]

Mudflap
[2005]

SSP
[2006]

_FORTIFY_SOURCE
[2005]

VTV
[2012]

Address Sanitizer
[2013]

Memory Sanitizer
[2013]

SafeStack
[2015]

IFCC
[2016]

適用にはソースコードが必要
(配布済みのバイナリに適用不可)

OS

Libsafe
[2000]

PaX
[2000]

W^X
[2003]

Exec Shield
[2003]

ASCII-Armor
[2003]

DEP
[2004]

ASLR
[2005]

vTPM
[2006]

SEHOP
[2007]

kASLR
[2013]

実行環境によっては
恩恵を受けられない

ハードウェア

TPM
[2004]

NX/XD bit
[2004]

Secure Boot
[2005]

Intel MPX
[2013]

研究目的

研究背景のまとめ

1. SBoF脆弱性は、現在も一定数報告されている
2. SBoF脆弱性を招くライブラリ関数を利用するバイナリが既に一定数配布されている
3. 既存の対策技術には、配布済みのバイナリに適用不可や実行環境に依存するといった問題がある



研究目的

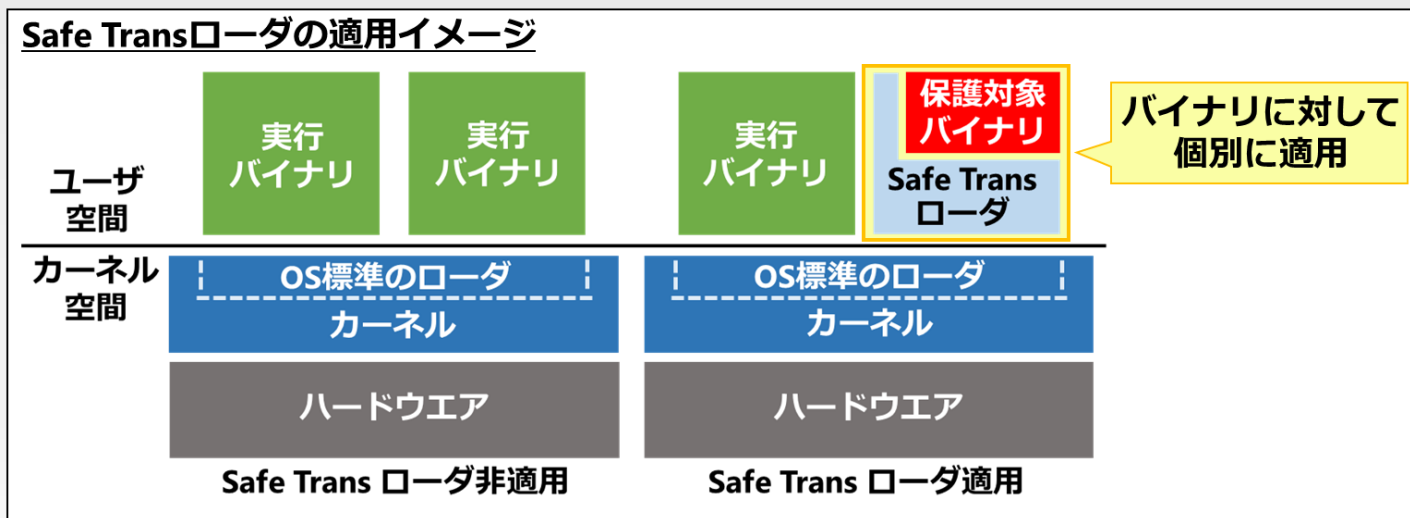
Safe Transローダ（後述）を用いて、
ライブラリ関数の置換によるSBoF攻撃の対策を実現する
（今回は、32bit LinuxOSにおけるELFバイナリの保護を想定）

提案手法(1/6)

Safe Transローダ

- ・アプリケーション層のプログラムローダ
- ・OS標準のローダに代わり保護対象バイナリをロードし、その際にバイナリ中で呼び出される**特定のライブラリ関数を他の関数に置換**

- ・今回は、SBoF脆弱性を招く **置換対象の関数** を **境界検査関数** に置換する



提案手法(2/6)

置換対象の関数と境界検査関数の一覧

2つの文献を参考に，14の関数を置換対象に選定

置換対象の関数	strcpy	strncpy	strcat	strncat	stpcpy
境界検査関数	Hstrcpy	Hstrncpy	Hstrcat	Hstrncat	Hstpcpy

置換対象の関数	memcpy	gets	getwd	realpath	sprintf
境界検査関数	Hmemcpy	Hgets	Hgetwd	Hrealpath	Hsprintf

置換対象の関数	snprintf	vsprintf	vsnprintf	scanf
境界検査関数	Hsnprintf	Hvsprintf	Hvsnprintf	Hscanf

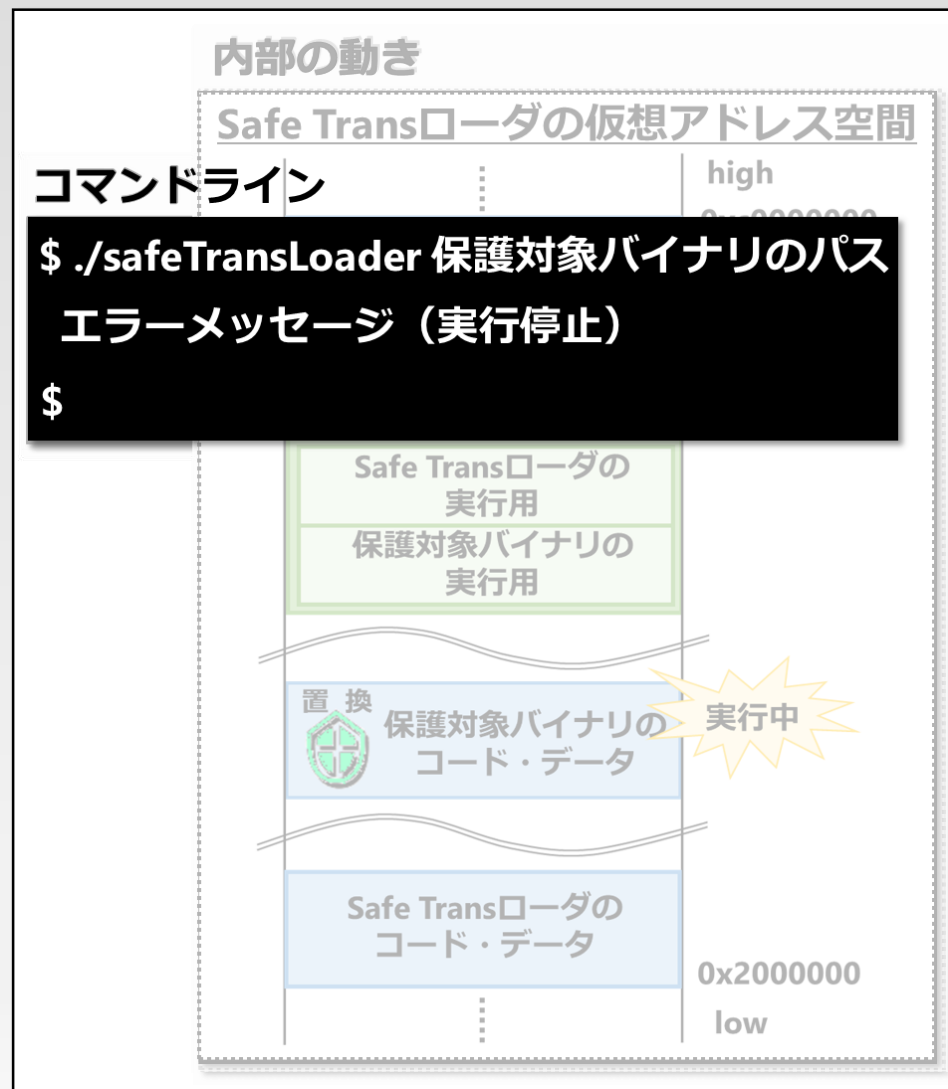
John Viega, Gary McGraw, 齋藤孝道, 河村政雄, 武舎広幸, 「Building Secure Software – ソフトウェアセキュリティについて開発者が知っているべきこと」, オーム社, 2006

Robert C. Seacord, 歌代和正, 久保正樹, 椎木孝斉,
「C/C++セキュアコーディング 第2版」, アスキー・メディアワークス, 2014

提案手法(3/6)

Safe Transローダの動作

1. コマンドラインから実行
2. Safe Transローダは、保護対象バイナリのコード部とデータ部をマップ
3. Safe Transローダは、保護対象バイナリに必要関数を置換し、再配置処理を行う
4. Safe Transローダは、保護対象バイナリのエントリポイントに制御を移す → 実行開始

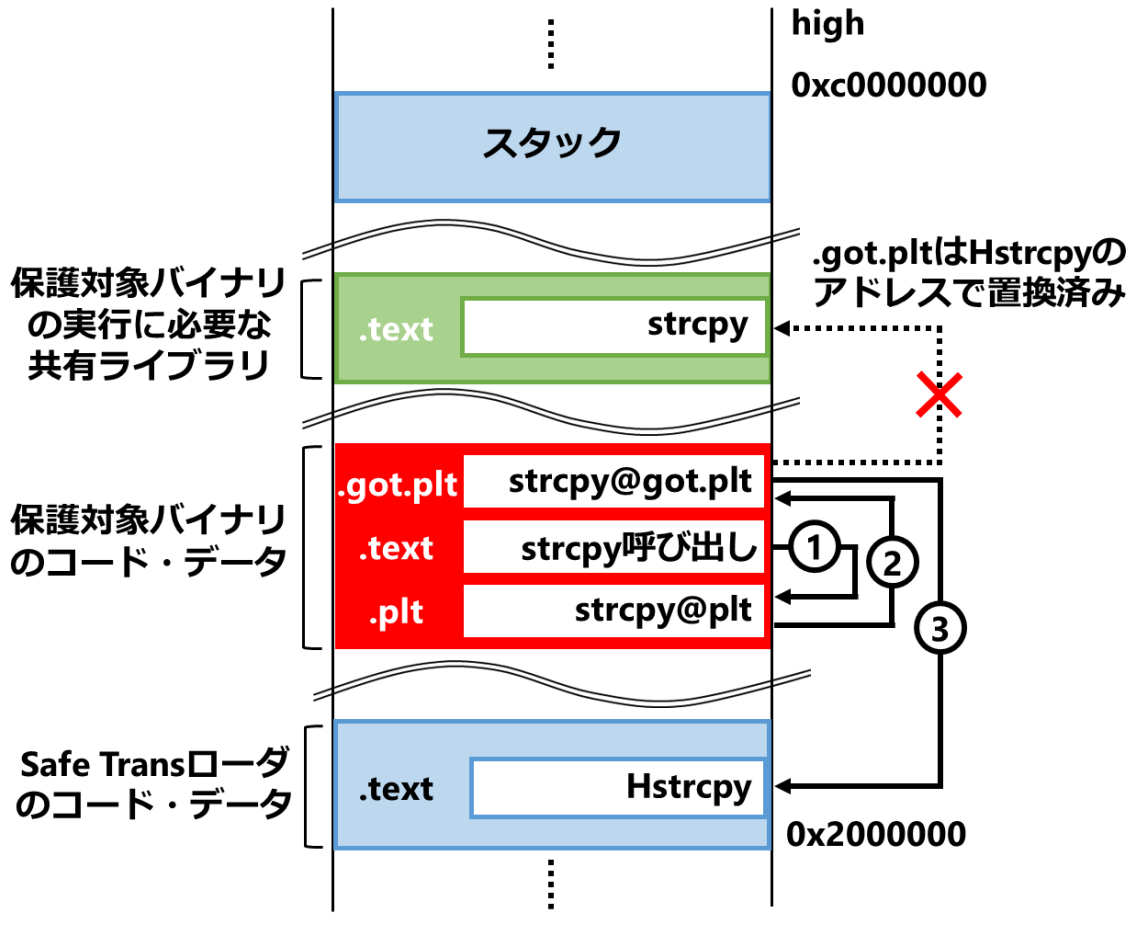


提案手法(4/6)

関数の置換方法

- ELFバイナリの実行において共有ライブラリ中の関数のアドレスは, **.got.plt** セクションに格納される
- Safe Transローダは, **.got.plt** セクションに境界検査関数のアドレスを書き込むことで関数を置換する

例) 関数の置換により, strcpy関数に代わって
Hstrcpy関数が呼び出される様子



提案手法(5/6)

境界検査関数

- Libsafeを参考に我々が独自に実装
- Safe Transロード上で定義されており，以下の処理でスタック上にある書き込み先のバッファの**境界検査を実現**
 1. 書き込み先のバッファの上限サイズを計算
 2. 書き込むデータサイズが計算した上限サイズ以下であれば，バッファへの書き込みを実行
(そうでなければ，プログラムの実行を停止)

**⇒ 境界検査を行うことで
書き込み先のバッファのオーバーフローを防ぐ**

提案手法(6/6)

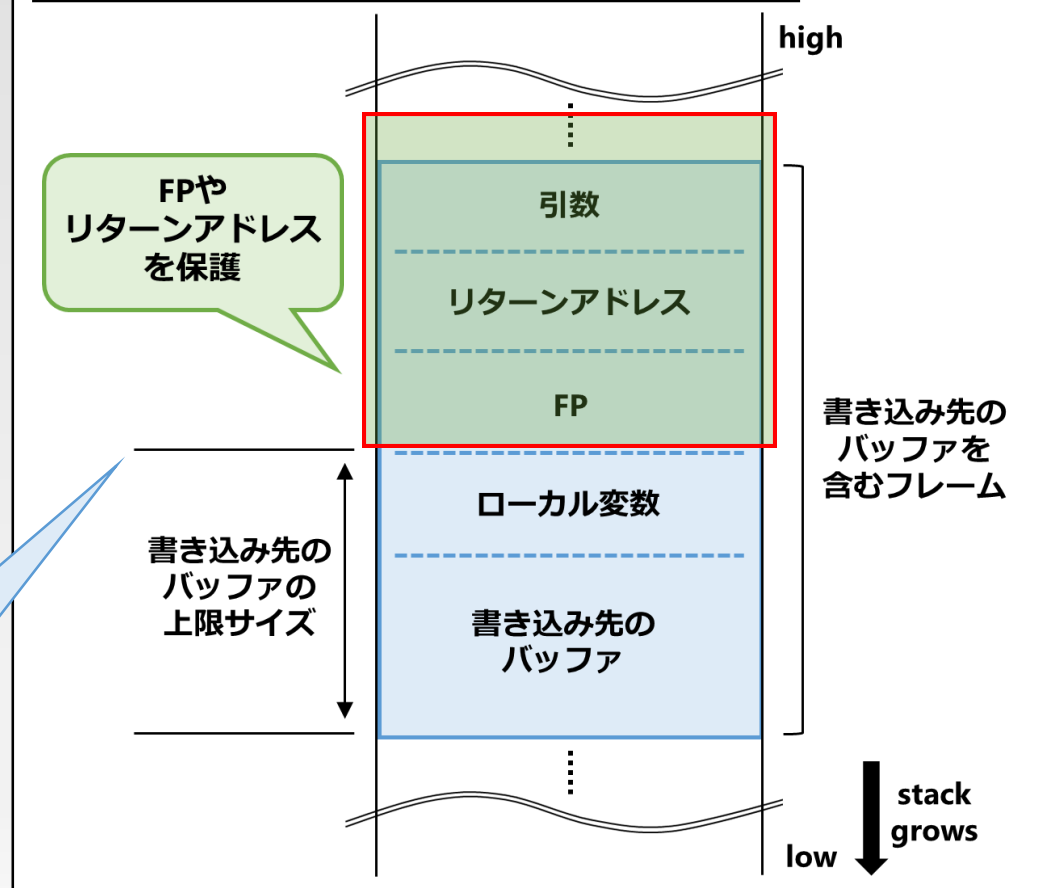
書き込み先のバッファの上限サイズの計算

書き込み先のバッファを含む
スタックフレームにおいて、
フレーム内のFPのアドレス
- バッファの先頭アドレス
として上限サイズを算出

※FP = フレームポインタ

このアドレスは、スタック
先頭のフレームからFPを
たどっていくことで取得

書き込み先のバッファの上限サイズ



評価(1/3)

評価方法

・有効性の評価

- － SBoF脆弱性を含む3種類のバイナリに対する
SBoF攻撃を防ぐことができるか検証

・パフォーマンスの評価

- － SPEC CPU2006によるベンチマーク結果を用いて,
Safe Transロード適用時のオーバーヘッドを計算

評価環境

CPU	Intel(R) Xeon(R) CPU E5620@2.40GHz
OS	Ubuntu14.04 LTS 32bit
GCC	4.8.4

評価(2/3)

有効性の評価

- ・ 評価の対象として以下の**3種**のバイナリを選定
 - [典型的なもの] CWE-121のページで公開されている
サンプルコードをもとに作成した**バイナリ2種**
 - [最近のもの] 実用バイナリである**Network Audio System1.9.3**
(CVE-2013-4256として脆弱性が報告されている)
- ・ 各バイナリをそれぞれSafe Transローダ上で実行し、
リターンアドレスを書き換える入力を与えた場合に
書き換えを防ぐことができるかを検証

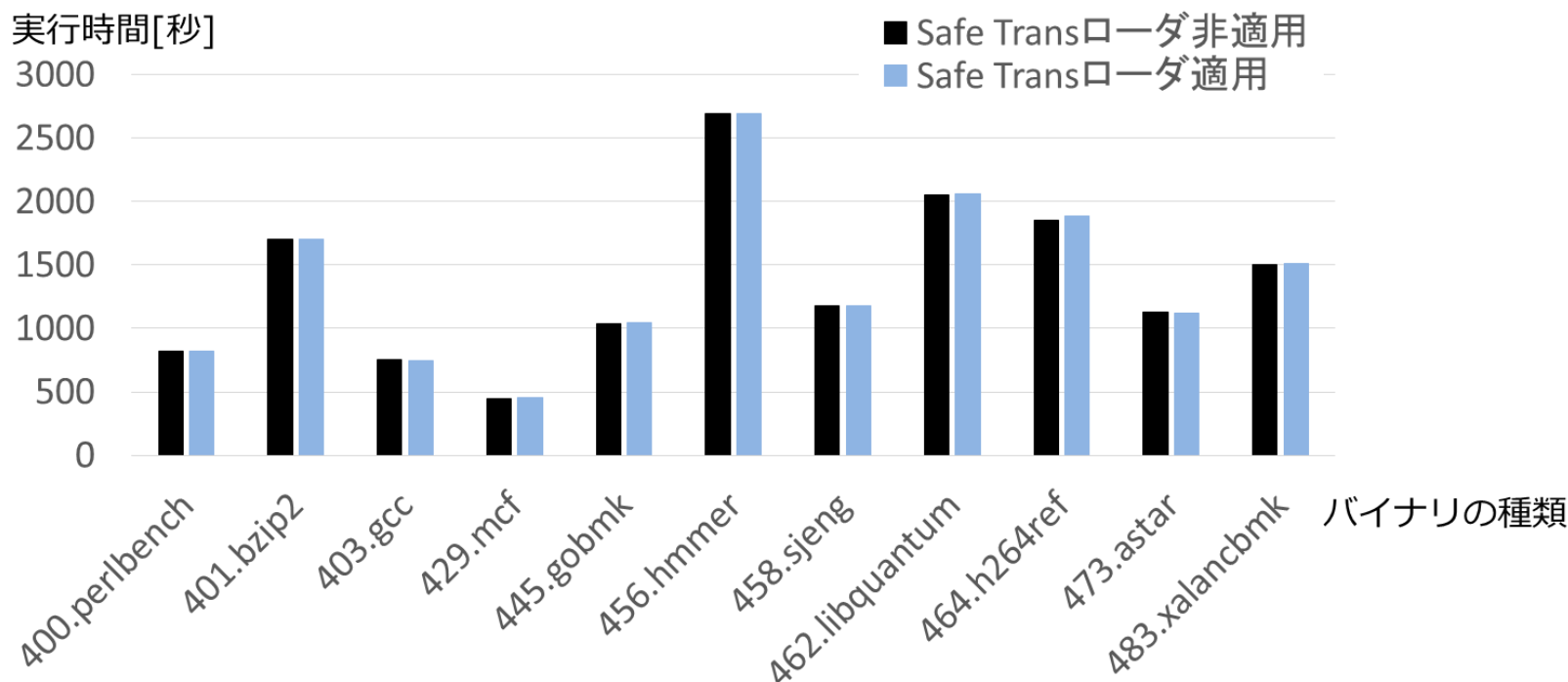
**3種すべてのバイナリに実行において、
リターンアドレスの書き換えを防ぐことを確認**

評価(3/3)

パフォーマンスの評価

- SPEC CPU2006を用いて, Safe Transローダ適用時と非適用時の実行時間を測定
- 平均オーバーヘッドは, **約0.087%**

測定結果のグラフ



ソフトウェアライフサイクルからみた考察

ライフ サイクル	開発フェーズ			運用フェーズ	
	設計	実装	テスト	実行	保守
対策技術 の分類	・コンパイル時対策 ・コーディング時対策			・セキュリティパッチ ・実行時対策 ← Safe Transロード	
適用者	開発者			利用者・運用者	

運用フェーズでの対策の重要性

開発者が脆弱性を作り込まないとは言い切れない
⇒ 開発フェーズで見逃された脆弱性の受け皿が必要

運用フェーズでの対策に求められること

1. ソースコードのないバイナリにも適用できること
2. 実行環境に手を加える必要がなく、利用者にとって適用が容易であること

本提案手法は ...
ロードの面で1を,
アプリケーションの
面で2を満たす
⇒ 運用フェーズでの
対策として有用

まとめ

- Safe Transローダは、**運用フェーズにおける対策技術**であり、ソースコードのないバイナリにも容易に適用できる
- Safe Transローダは、特定のライブラリ関数を利用するバイナリに対するリターンアドレスを書き換えるようなSBoF攻撃に有効
- Safe Transローダ非適用時に対する適用時のオーバーヘッドは、**約0.087%**と非常に小さい

今後の課題

- ・ **境界検査関数の改善点**

- バッファとフレームポインタの間に配置されたローカル変数の書き換えも防げるように

- ・ **Safe Transローダ自体の改善点**

- C++の例外処理を含むプログラムの実行への対応

Safe Transローダの課題：

C++の例外処理（DWARFを用いた方式）
について

C++の例外処理

- 例外オブジェクトをthrowすると、そのオブジェクトの型に対応するcatchブロックが見つかるまで、呼び出し元の関数に遡りながら探索する
 - 関数呼び出し前のスタックやレジスタの状態の復元が必要

大域脱出 (Unwind) と呼ばれる

- GCCが提供する大域脱出の方式
 - Unwind-Sjlj方式 (setjmp/longjmp関数を利用)
 - Unwind-dw2方式
(バイナリ中に埋め込んだDWARF2形式のデバッグ情報を利用)

Safe Transローダは、Unwind-dw2方式で
例外処理を行うプログラムを正常に実行できない

バイナリ中のデバッグ情報

- GCCが生成するELFバイナリには,
 - **.eh_frame**セクション
 - **.eh_frame_hdr**セクション (= **GNU_EH_FRAME**セグメント)という形で, デバッグ情報が埋め込まれる
- **.eh_frame**セクションは, 複数の FDE のリストで構成される
 - **Frame Description Entry**
 - 関数 (アドレス〇〇から××の範囲の命令) ごとに, スタックやレジスタがどの程度変化するかを記録したもの
- **.eh_frame_hdr**セクションは, **.eh_frame**セクション内の FDE の中で, 現在の eip の値に対応するものを探索しやすくするためのセクション
 - 例外処理時は, **.eh_frame_hdr** → **.eh_frame**とたどって, FDEを取得することでUnwindする

例外処理プログラムの の実行結果

実行環境

OS	...	Ubuntu14.04 32bit版
カーネル	...	3.13.0-24-generic
gcc / glibc	...	4.8.2 / 2.19

exception.cpp

```
#include <iostream>
#include <cstdio>
using namespace std;

int main() {
    try {
        const char* str = "exception test";
        puts("before throw");
        throw str;
    }
    catch (const char *str) {
        puts(str);
    }

    return 0;
}
```

ビルド (Unwind-dw2方式で生成される)

```
$ g++ exception.cpp
-g -o exception
```

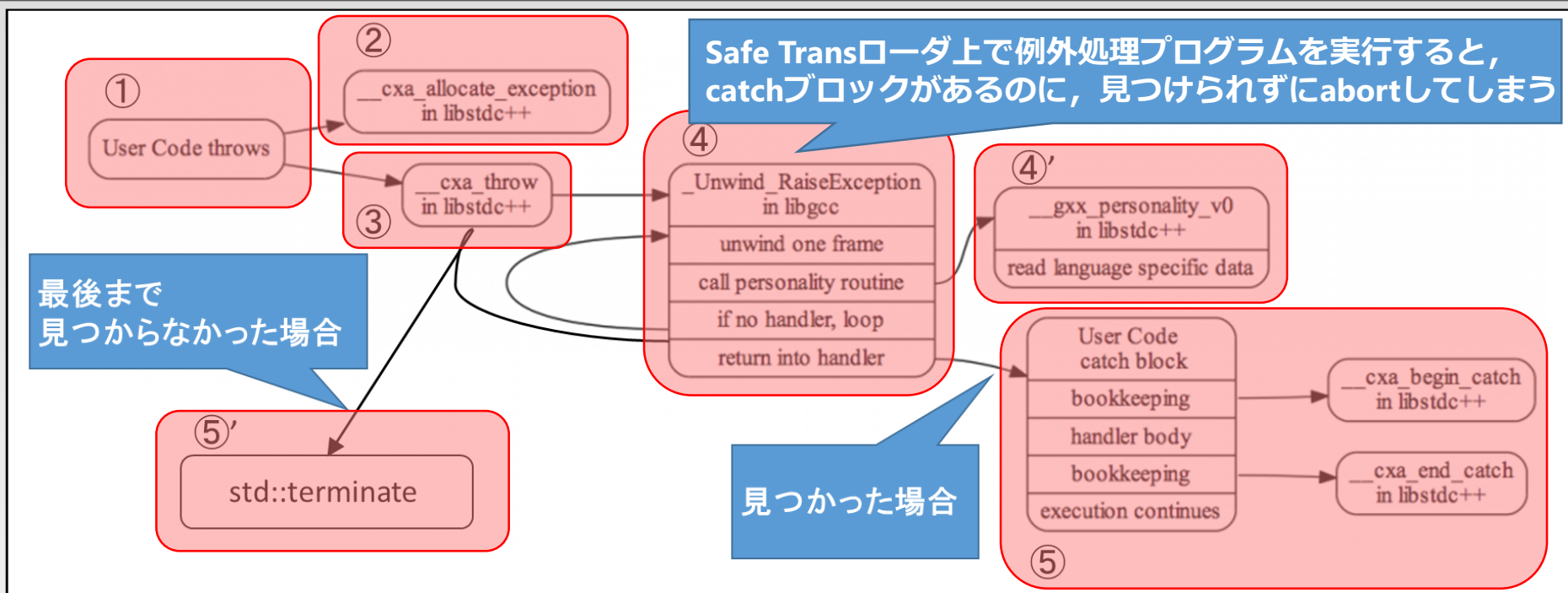
実行結果 (Safe Transローダ非適用)

```
$ ./exception
before throw
exception test
```

実行結果 (Safe Transローダ適用)

```
$ ./safeTransLoader exception
before throw
terminate called after throwing
an instance of 'char const*'
Program received
signal SIGABRT, Aborted.
```


例外発生時の処理のフロー



- ① 例外発生
- ② 例外オブジェクトの生成
- ③ オブジェクトのthrow
- ④ FDEの取得とUnwind

- ④' catchブロックが存在するかチェック
- ⑤ catchブロックの実行
- ⑤' プログラムのabort

_Unwind_RaiseException (1)

内部で以下のように関数を呼び出す

uw_init_context_1
→ uw_frame_state_for
→ _Unwind_Find_FDE
→ **dl_iterate_phdr**

- 実行中のプロセスから動的リンクされている共有ライブラリをたどっていき、引数で指定した **_Unwind_IteratePhdrCallback** を実行
- **_Unwind_IteratePhdrCallback**で、実行中のプロセスの**GNU_EH_FRAME**セグメント(= **.eh_frame_hdr**セクション)を探索

しかし、Safe Transローダ上での実行の場合、
実行中のプロセスは、あくまでSafe Transローダとみなされる
→ **保護対象バイナリではなく、Safe Transローダ自身のGNU_EH_FRAMEセグメントに基づいてUnwindしてしまうのでは？**

dl_iterate_phdrする プログラムの実行結果 (1)

iterate.c ... 実行中のプロセスと動的リンクしている共有ライブラリをたどり,
callbackによりプログラムヘッダ中のセグメント情報を入力するプログラム

```
#define _GNU_SOURCE
#include <link.h>
#include <stdlib.h>
#include <stdio.h>

static int callback(struct dl_phdr_info *info, size_t size, void *data) {
    int j;
    printf("name=%s (%d segments)%n", info->dlpi_name, info->dlpi_phnum);
    for (j = 0; j < info->dlpi_phnum; j++)
        printf("%t%t header %2d: address=%10p%n", j,
            (void *) (info->dlpi_addr + info->dlpi_phdr[j].p_vaddr));
    return 0;
}

int main(int argc, char *argv[]) {
    dl_iterate_phdr(callback, NULL);
    exit(EXIT_SUCCESS);
}
```

dl_iterate_phdrする プログラムの実行結果 (2)

実行結果 (Safe Transロード**非適用**)

```
$ ./iterate
name= (9 segments)
  header 0: address= 0x8048034
  header 1: address= 0x8048154
  header 2: address= 0x8048000
  header 3: address= 0x8049f08
  header 4: address= 0x8049f14
  ...
```

Safe Transロード非適用の場合、
iterateのセグメント情報が表示
される

```
header 0: address=0xb75c3034
...
name=/lib/ld-linux.so.2 (7 segments)
  header 0: address=0xb7789000
  ...
```

実行結果 (Safe Transロード**適用**)

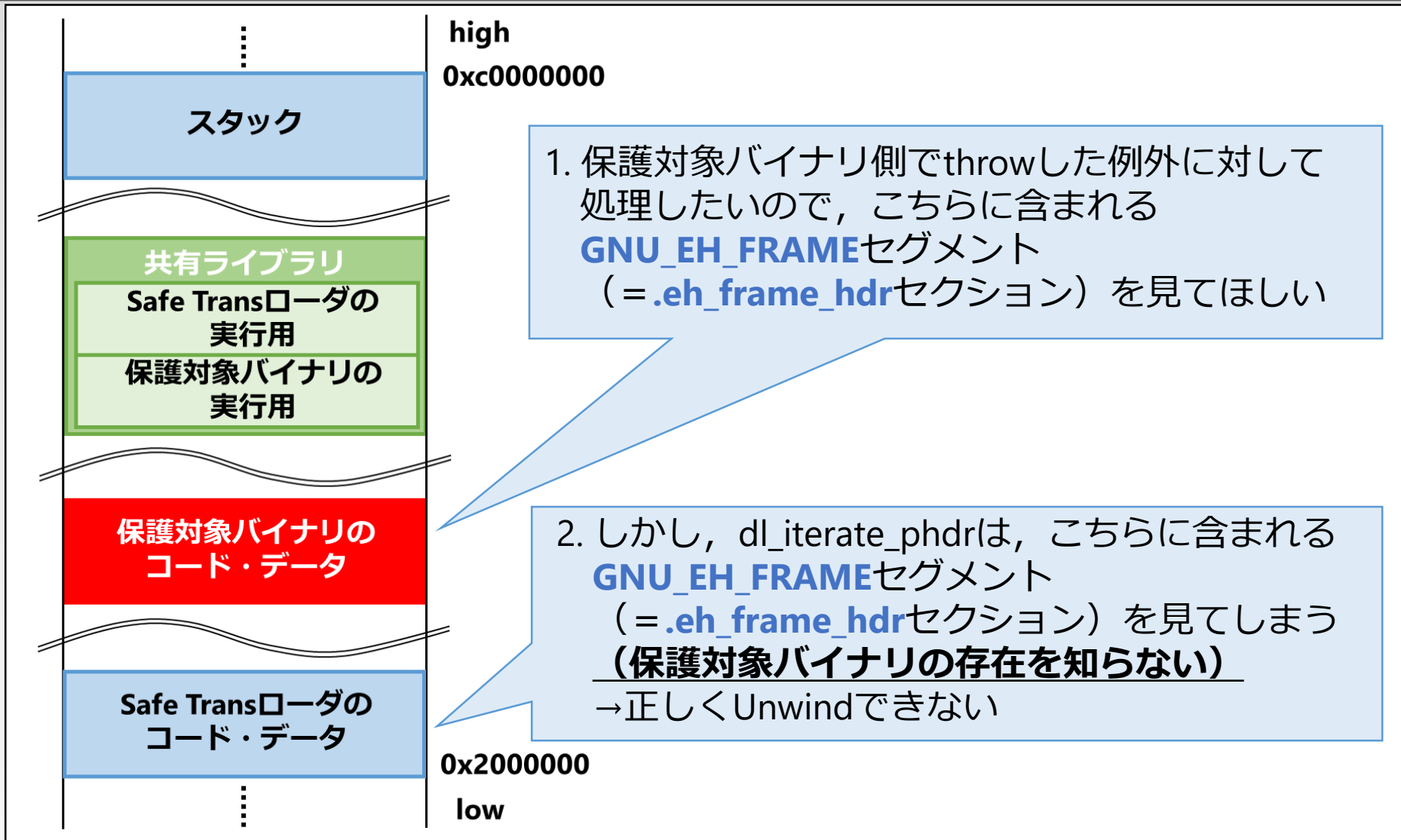
```
$ ./safeTransLoader iterate
name= (9 segments)
  header 0: address= 0x2000034
  header 1: address= 0x2000154
  header 2: address= 0x2000000
  header 3: address= 0x2003f00
  header 4: address= 0x2003f0c
  ...
```

Safe Transロードを適用した場合、
iterateではなく、Safe Transロード
のセグメント情報が表示される

→ 実行中のプロセスは、あくまで
Safe Transロードとみなされる

```
...
name=/lib/ld-linux.so.2 (7 segments)
  ...
```

_Unwind_RaiseException (2)



お聞きしたいこと

- unwind-dw2方式の例外処理の詳しい動作
または、デバッグで追う際のコツ
- **.eh_frame**セクションや**.eh_frame_hdr**セクション中のバイナリデータは、どのように解釈されるのか
- dl_iterate_phdr関数は、どのデータ構造を参照して、
実行中のプロセスや共有ライブラリをたどっているのか
- Safe Transローダ上で例外処理プログラムを
実行できない他の原因がありそうか

